Les bugs vicieux du multithread

Développer des services Windows n'a jamais été facile. Pour simplifier, un service Windows est une application qui contient 3 points d'entrées du style OnStart(), OnStop() et OnPause().

Un service se démarre avec « net start mon_service » ou bien dans le gestionnaire de services par un bouton droit « Démarrer le service ». L'arrêt ou la pause sont sur le même principe. Un service a pour vocation de réaliser des tâches en arrière-plan, comme par exemple :

- Attentes de connexions client entrantes (tcp/ip, web services, etc.) ;
- Lancement de calculs ou de tâches longues ;
- Traitement batchs divers ;
- Gestionnaire de ressources diverses (cache de données).

Pour réaliser les opérations en tâches de fond, un service crée et gère des threads. Un thread se code comme une routine qui peut prendre des paramètres. Le système d'exploitation assure l'exécution des threads en parallèle et en prenant soin de dispatcher, plus ou moins finement, les threads sur les différents processeurs disponibles. L'implémentation suivant le langage et les librairies utilisées ne garantit pas le même niveau de montée en charge et d'exploitation des ressources CPU disponibles. Mais ceci fera l'objet d'un autre article. La difficulté de la mise au point de ce genre d'applications réside dans le fait que les opérations assurées par un service peuvent dépendre du contexte d'exécution. En effet, dans le cadre des services « communiquant », il faut qu'un client cherche à communiquer avec le service pour déclencher un certains nombres d'opérations. Dans le cadre de services « batchs », les traitements peuvent durer longtemps. La mise au point de ce type de programme requiert une certaine rigueur afin d'éviter d'être confronté aux situations suivantes :

- Le service plante au bout de 10 minutes…
- Le service plante aléatoirement…
- Le service plante dès que 2 clients se connectent…
- Le service plante et je n'ai pas d'erreur dans le fichier de traces…

Ce genre de situation est délicat à aborder car les pistes d'investigation sont, au début, très minces. Comment éviter cette situation ?

PRENDRE DES PRÉCAUTIONS

Avant tout, il faut permettre le debugging du service comme n'importe quelle application. Il faut être capable de le faire tourner en mode DEBUG et tirer parti des informations remontées par le debugger pour trouver les différentes erreurs. La première chose est de rendre le service identique à une application console. Je dois pouvoir lancer le service sans la capsule « Windows Service ». Si je peux lancer « mon_service.exe –console », alors mon debugger pourra travailler. Sinon, il faut que je lance le service manuellement que j'attache le debugger au ProcessId du service…. C'est pénible et long car chaque lancement donnera un ProcessId différent…

Ensuite, il faut utiliser un framework de gestion des traces qui possède un mode « verbose ». Je dois pouvoir au travers du fichier de config passer en mode DEBUG, INFO ou ERROR. Dans le mode ERROR, seules les traces de type erreurs sont visibles. Dans le mode INFO, seules les opérations importantes et les erreurs sont visibles. Dans le mode DEBUG, toutes les traces sont visibles. Cela veut dire que le programme doit comporter une classe de trace qui ressemble de près ou de loin à cela :

```
enum ModeTraceEnum { Error, Info, Debug };

class CMyTrace
{
public:
    static void Log (ModeTraceEnum mode, std::string message);
    ...
};
```

Il existe de nombreux frameworks de traces qui feront le job correctement et ce pour différents langages (log4cxx, log4net, entlib, etc.). Cela veut donc dire que lors de l'écriture du code d'un service, le ou les développeurs doivent faire l'effort de rendre l'application communicante. Ainsi en cas de problèmes ou de doutes, on fait tourner le service en mode DEBUG et l'analyse des traces permet d'avoir un indice sur le traitement qui est en erreur. La méthode Bubule() n'a pas terminé, la méthode Totor() n'est jamais appelée, etc.

LA GESTION DES DONNÉES

La gestion des structures de données en mémoire est souvent la cause de plantage des applications. Tous les langages possèdent des collections de type vector, list, map, array et dans la plupart des cas, ces classes ne sont pas thread-safe ! Et oui, il est possible de lire les données en parallèle mais si un thread réalise une écriture alors, toutes les opérations de lecture doivent être protégées car sinon, on s'expose à un plantage potentiel. Dans la pratique tout va bien tant que votre application est un bloc mono-thread mais le jour ou vous y ajoutez des threads, par exemple pour surveiller le statut de certaines opérations, et bien vous devez alors protéger l'ensemble des variables communes… Dans ce cas, vous entrez dans le monde merveilleux de la synchronisation et de nouvelles classes sont disponibles : Mutex, Event, CriticalSection, Semaphore et un tas d'autres moyens évolués pour utiliser ce qui va se rapprocher de près ou de loin de l'utilisation de méthodes comme Lock() et Unlock(). Pour améliorer la performance des applications, il faut aller au-delà d'un mécanisme de verrou classique et utiliser des verrous qui permettent de lire en parallèle et de protéger en écriture : c'est le « Single Writer Multiple Readers Guard ». Ce type de verrou est plus efficace mais demande aussi un code plus fin car il faut indiquer le type d'opération (lecture ou écriture).

Ainsi pour lire une donnée, on code :

```
        g_Guard.WaitToRead();
pObjectFound = _my_data.GetFirst();
strPasswordCheck = pObjectFound->m_strPassword;
g_Guard.Done();
```

Et pour écrire une donnée, on code :

```
g_Guard.WaitToWrite();
_my_data.InsertAt(0, pNewObject );
g_Guard.Done();
```

Ce type de code possède toutes les garanties de protection des variables partagées et cependant cela n'évitera peut-être pas une recherche longue et fastidieuse pour trouver un bug sérieux. Dans certains cas, même avec un bon niveau de log, des verrous et une gestion d'erreur bien réalisée, on peut passer beaucoup de temps sur un plantage aléatoire et là, il faut utiliser une autre technique.

UTILISATION D'UN DEBUGGER SYSTÈME

Comment trouver des indices sur un programme qui ne contient pas de traces exploitables… Ce cas est plus fréquent qu'on ne pourrait le penser. Le plantage de l'application se traduit par une boîte de dialogue générique qui ressemble à : [Fig.1]. Dans ce cas, celui qui cherche à avoir des indices peut se servir du système sur lequel tourne son application. Attacher un debugger à une application va permettre de mettre l'application sous surveillance. Si un bout de code provoque une erreur, le système va capturer les erreurs et donner des indications :

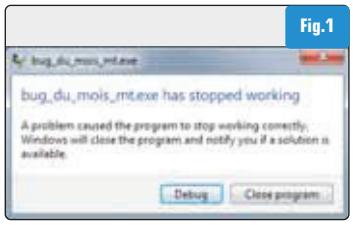

Fig.1

- Nom du module en erreur (ex : Foo.dll)
- Nom de la routine qui provoque le plantage (ex : ReadFile)
- Pile des derniers appels (ex : ReadFile(), RoutineProcessAFile(), RoutineBatchNum102(), OnStartService())

Cependant dans la pratique, la pile des derniers appels (call stack) est souvent difficile à lire car elle fait référence à des routines systèmes de l'OS ou soit à des routines des librairies de bas niveau comme kernel32 ou msvcrxx.dll. De plus, pour être capable d'afficher une pile lisible, les debugger cherchent à tirer parti de « symboles de debug ». Et pourtant, il ne faudrait pas beaucoup d'efforts pour arriver à obtenir une fenêtre qui donne les mêmes indications qu'une fenêtre de debugging de Visual Studio comme celle-ci : [Fig.2]

La routine qui a provoquée l'erreur se nomme « CFoo::RoutineDeFolie » et l'erreur est arrivée en ligne 75. Et voila, il ne reste plus qu'à corriger la ligne en question. Pour arriver à un schéma aussi confortable, il faut déposer sur la machine ou tourne l'application les informations symboliques de debugging. Oui ? Et c'est quoi ???

Et bien, lors de la phase de compilation du projet, il n'y a pas que les modules EXE ou DLL qui sont produits. Il y a aussi des fichiers PDB et DBG. Ceci est valable en mode Release ou en mode Debug. Ces

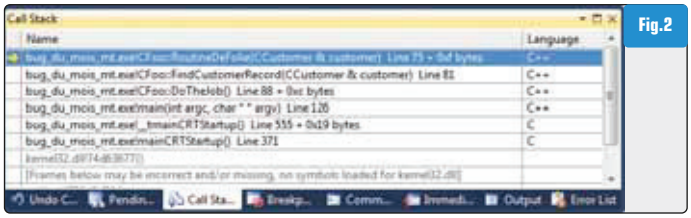

Fig.2

fichiers contiennent les diverses informations permettant à un debugger de retrouver son chemin avec la liste des symboles (fonctions, variables, nom de fichier source, numéro de ligne, etc.). Ainsi dans notre exemple, si on lance un debugger comme WinDbg, on ouvre l'exécutable, on le fait tourner et on attend que cela finisse mal… Cela va se traduire par un message de la sorte :

```
(1e7c.c64): Break instruction exception - code 80000003 (first chance)
```

Le debugger arrête le programme et on a la possibilité de demander la dernière liste d'appels avec le crash. Et là, on remarque que l'information est bien présente : c'est bien la routine « CFoo::RoutineDeFolie » qui est mise en cause.

```
0034fa38 01242566 bug_du_mois_mt!CFoo::RoutineDeFolie+0x78
[ .. bug_du_mois_mt.cpp @ 75]
```

[Fig.3]

CONCLUSION

La mise au point d'application comme les services Windows où les applications multithreads n'est pas très simple mais l'utilisation de traces, des verrous évolués et des symboles de debugging facilitent grandement la tâche. Le point fort de cette méthode de debugging « post-mortem » est que je peux avoir les informations sur un plantage d'application sur une machine qui ne possède pas d'environnement de développement, ni les sources de l'application. C'est le cas des environnements de production. Il faut juste publier les informations symboliques PDB et DBG pour chaque module (EXE ou DLL) et installer le debugger WinDbg. WinDbg est disponible dans le Platform SDK sous « %Program Files%\Microsoft SDKs\Windows\v7.1\Redist\Debugging Tools for Windows» en version x86 et x64, il se nomme dbg_x86.msi ou dbg_x64.msi et fait 20 MB environ.

Christophe **Pichaud** - *.NET Rangers by Sogeti*
Consultant sur les technologies Microsoft
christophepichaud@hotmail.com - www.windowscpp.net

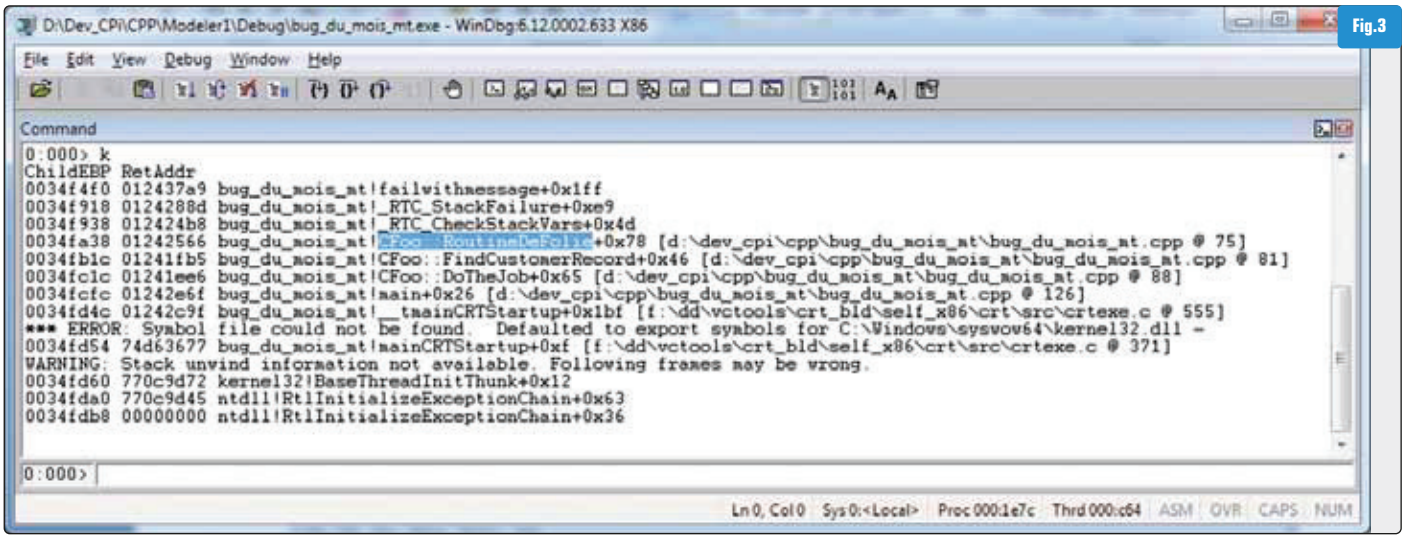

Fig.3